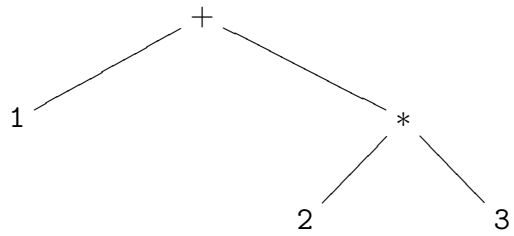
5. Wykład 5: Wprowadzenie do rachunku λ.

Na dzisiejszym wykładzie zajmiemy się rachunkiem λ, czyli podstawowym rachunkiem wykorzystywanym przez języki funkcyjne, w tym oczywiście przez SML. Dzięki niemu będziemy potrafili uchwycić najważniejsze mechanizmy działania języków funkcyjnych, a zatem zrozumieć podstawy teorertyczne ich funkcjonowania. Zgodnie z tezamu Churcha-Turinga (o których potem), siła wyrazu rachunku λ jest tak samo duża, jak abstrakcyjnych maszyn Turinga, wszelako jego syntaks jest istotnie łatwiejszy. Na początek zajmiemy się przedyskutowaniem syntaksu rachunku λ, a potem pokażemy, w jaki sposób można w nim pisać programy.

Zanim zaczniemy, omówimy pokrótce różnicę między **syntaksem abstrakcyjnym** i **syntaksem konkretnym.** Syntaks konkretny specyfikuje który ciąg znaków zostanie zaakceptowany jako prawidłowo napisany program (a więc program nie wywołujący błedów syntaksu), a który zostanie odrzucony jako program nieprawidłowy (a więc zgłaszający błędy syntaksu). Na przykład, zgodnie z konkretnym syntaksem SML, ciąg znaków `~1` zostanie zinterpretowany jako liczba całkowita -1, natomiast ciąg znaków `-1` zostanie zinterpretowany jako operator $-$ zastosowany do argumentu całkowitego `1` (co w dalszej kolejności doprowadzi do błędu). Parser implementujący konkretny syntaks zwykle tłumaczy kod źródłowy na odpowiednie drzewka. Na przykład, kod źródłowy `1 + 2 * 3` będzie przetłumaczony na:



oczywiście po wcześniejszym uwzględnieniu reguł pierwszeństwa operatorów. Tego typu drzewka nazywać będziemy **drzewami abstrakcyjnego syntaksu,** które powstają niezależnie od szczegółów parsowania (takich jak reguły łączności czy pierwszeństwa operatorów) i skupiają się na strukturze kodów źródłowych; abstrakcyjny syntaks jest po prostu syntaksem dla takich struktur drzewek.

Jakkolwiek konkretny syntaks jest nieodzowną częścią projektowania języka programowania, nie będziemy się nim wcale zajmować na tym wykładzie. Zamiast tego skupimy się całkowicie na abstrakcyjnym syntaksie i skoncentrujemy się na aspektach rachunkowych projektowania języków programowania. Na przykład, nie będziemy zbytnio zastanawiali się, dlaczego `1 + 2 * 3` oraz `1 + (2 * 3)` są zamieniane przez parser na taki sam abstrakcyjny syntaks jak pokazany na drzewku powyżej. Celem zrozumienia w jaki sposób obliczamy wartość każdego z powyższych wyrażeń, w zupełności wystarczy ograniczyć się do rozważania abstrakcyjnego syntaksu.

5.1. Abstrakcyjny syntaks rachunku λ.

Abstrakcyjny syntaks dla rachunku λ dany jest następująco:

$$\text{wyraenie} \qquad e \quad ::= \quad x | \lambda x.e || ee$$

- Wyrażenie x jest nazywane **zmienną**. Możemy oczywiście użyć innych nazw dla zmiennych, na przykład z, s, t, f, arg, $accum$ itd. Mówiąc ściśle, x jest metazmienną w indukcyjnej definicji wyrażenia.
- Wyrażenie $\lambda x.e$ jest nazywane **λ-abstrakcją,** albo po prostu funkcją, która oznacza funkcję matematyczną, której formalnym argumentem jest x, a której zawartość zadana jest przez e. Możemy myśleć o $\lambda x.e$ jak o wewnętrznej reprezentacji funkcji w SML bez nazwy danej przez `fn` x `=>` e w abstrakcyjnym syntaksie. Mówimy, że zmienna x jest **związana** w λ-abstrakcji $\lambda x.e$ (dokładnie tak, jak zmienna x jest związana w funkcji SML `fn` x `=>` e).

- Wyrażenie e_1e_2 jest nazywane **λ-aplikacją** lub po prostu **aplikacją,** która oznacza zastosowanie funkcji (o ile możemy w jakiś sposób pokazać, że e_1 jest równoważne pewnej λ-abstrakcji). Możemy myśleć o e_1e_2 jak o wewnętrznej reprezentacji zastosowania funkcji w SML w abstrakcyjnym syntaksie. Tak jak w SML, aplikacje są łącznej z lewej strony, a zatem $e_1e_2e_3$ czytamy jak $(e_1e_2)e_3$, a nie $e_1(e_2e_3)$.

Zakres λ-abstrakcji rozciąga się tak daleko w prawo, jak tylko jest to możliwe. Oto garść przykładów:

- $\lambda x.xy$ jest takim samym wyrażeniem, jak λ-abstrakcja $\lambda x.(xy)$, której zawartością jest xy. Nie należy jej rozumieć jak aplikacji $(\lambda x.x)y$.
- $\lambda x.\lambda y.xy$ jest takim samym wyrażeniem, jak $\lambda x.\lambda y.(xy) = \lambda x.(\lambda y.(xy))$. Nie należy jej rozumieć jak aplikacji $(\lambda x.\lambda y.x)y$.

Jak się okazuje, każde wyrażenie w rachunku λ oznacza pewną funkcję matematyczną. Dokładniej, oznaczenie każdego wyrażenia w rachunku λ jest funkcją matematyczną. Poniżej zobaczymy, w jaki sposób można określić jednoznacznie wyznaczone funkcje matematyczne odpowiadające wyrażeniom w rachunku λ, obecnie zaś spróbujemy wyrobić sobie odpowiednią intuicję poprzez rozważenie kilku przykladów λ-abstrakcji.

Naszym pierwszym przykładem niech będzie funkcja identycznościowa:

$$\mathsf{id} \quad = \quad \lambda x.x$$

id jest funkcją identycznościową, albowiem przy danym argumencie x zwraca wartość x bez żadnych dodatkowych obliczeń. Tak jak funkcje wyższego rzędu w SML, λ-abstrakcje mogą zwracać inne λ-abstrakcje jako wartości. Na przykład, tt zdefiniowane poniżej argumentowi t przyporządkowuje λ-abstrakcję $\lambda f.t$, która ignoruje swój własny argument; ff z kolei ignoruje swój własny argument i przyporządkowuje λ-abstrakcję $\lambda f.f$:

$$\mathsf{tt} \quad = \quad \lambda t.\lambda f.t = \lambda t.(\lambda f.t)$$
$$\mathsf{tt} \quad = \quad \lambda t.\lambda f.f = \lambda t.(\lambda f.f)$$

Podobnie λ-abstrakcja może przyjmować inną λ-abstrakcję jako argument. Na przykład, poniższa λ-abstrakcja przyjmuje inną λ-abstrakcję s, która następnie jest stosowana do z:

$$\mathsf{one} \quad = \quad \lambda s.\lambda z.sz = \lambda s.(\lambda z.(sz)).$$

5.2. **Semantyka operacyjna rachunku λ.** Semantyka danego języka programowania odpowiada na pytanie, jakie jest "znaczenie" danego programu. Jest to ważna kwestia przy projektowaniu oprogramowania, albowiem brak formalnej semantyki może skutkować potencjalną niejednoznacznością w działaniu interpretera. Mówiąc wprost, brak formalnej semantyki sprawia, że odczytanie znaczenia pewnych programów staje się całkowicie niemożliwe. Żeby było ciekawiej, nie każdy język ma zdefiniowaną formalną semantykę. Na przykład (co może by dla Was zaskoczeniem) język C jest pozbawiony formalnej semantyki – w rezultacie ten sam program w języku C może różnie się zachowywać w zależności od stanu, w jakim znajduje się urządzenie, na którym jest uruchamiany.

Istnieją trzy podejścia do formułowania semantyki języków programowania: **semantyka opisowa, semantyka aksjomatyczna** i **semantyka operacyjna.** Na tym wykładzie zajmiemy się wyłącznie semantyką operacyjną, ze względu na jej bliski związek z osądami i regułami wnioskowania. Podejście to ma też tę zaletę, że w bezpośredni sposób odzwierciedla implementację języków programowania (a więc interpretery i kompilatory).

Ogólnie rzecz biorąc, semantyka operacyjna języka programowania specyfikuje w jaki sposób można zamienić program na **wartość** poprzez ciąg "operacji". W przypadku rachunku λ, wartości składają się z λ-abstrakcji, natomiast "operacje" zwane są **redukcjami**. Tym samym semantyka operacyjna

rachunku λ określa w jaki sposób można zredukować wyrażenie e do wartości v, gdzie v zdefiniowane jest następująco:

$$\mathsf{value} \quad v \quad ::= \quad \lambda x.e$$

Wartość v będziemy rozumieli jako znaczenie e. Jako że λ-abstrakcja oznacza funkcję matematyczną wynika stąd, że **każde** wyrażenie w rachunku λ oznacza funkcję matematyczną.

Pamiętając o ogólnej idei, jaka nam przyświeca, zdefiniujmy teraz formalnie redukcje wyrażeń. Wprowadźmy mianowicie **osąd redukcji** postaci $e \mapsto e'$ następująco:

$$e \mapsto e' \quad \Leftrightarrow \quad e \text{ redukuje się do } e'$$

Oznaczmy przez $\mapsto^*$ domknięcie przechodnio-zwrotne relacji $\mapsto$ (a więc najmniejszą relację przechodnią i zwrotną, zawierającą $\mapsto$; nietrudno sprawdzić, że $e \mapsto^* e'$ zachodzi wtedy i tylko wtedy, gdy dla pewnych $e_1, \ldots, e_n$ zachodzi $e \mapsto e_1 \mapsto \ldots \mapsto e_n \mapsto e'$. Będziemy mówili, że e **przyjmuje wartość** v, gdy zachodzi $e \mapsto^* v$.

Zanim przejdziemy do podania reguł wnioskowania, zastanówmy się, jakiego rodzaju wyrażenia powinny dać się redukować do innych wyrażeń. Oczywiście zmienne i λ-abstrakcje nie powinny być redukowalne:

$$x \not\mapsto .$$
$$\lambda x.e \not\mapsto .$$

(symbol $e \not\mapsto .$ oznacza, że e nie redukuje się do innego wyrażenia). Kiedy powinniśmy być w stanie zredukować aplikację $e_1 e_2$? Jeżeli będziemy myśleć o niej jak o wewnętrznej reprezentacji aplikacji funkcji SML, możemy ją zredukować tylko wtedy, gdy e_1 reprezentuje funkcję SML. Tym samym jedyny kandydat na redukowalną aplikację jest postaci $(\lambda x.e_1')e_2$.

Jeżeli będziemy myśleć o $\lambda x.e_1'$ jak o funkcji matematycznej o argumencie x i zawartości e_1', najbardziej naturalnym sposobem redukcji $(\lambda x.e_1')e_2$ jest podstawienie e_2 w każdym miejscu występowania e_1' lub, równoważnie, przez zastąpienie każdego wystąpienia x w e_1' przez e_2 (od tego momentu nie będziemy się przejmować tym, czy e_2 jest wartością, czy nie). Wobec tego możemy wprowadzić **podstawienie** $[e'/x]e$ następująco:

$[e'/x]e$ jest zdefiniowane jako wyrażenie powstałe przez zastąpienie każdego wystąpienia x w e przez e'

$[e'/x]e$ może być również odczytywane jako "zastosowanie podstawienia $[e'/x]$ do e." Uzasadnia to następującą redukcję:

$$(\lambda x.e)e' \mapsto [e'/x]e$$

gdzie redukowane wyrażenie, a mianowicie $(\lambda x.e)e'$, zwane jest **redeksem**. Ze względów historycznych, powyższa redukcja zwana jest β-redukcją.

Jakkolwiek to co zauważyliśmy powyżej może wydawać się proste, precyzyjna definicja podstawienia $[e'/x]e$ jest o wiele subtelniejsza i zajmiemy się nią bardziej szczegółowo na kolejnym wykładzie, ograniczając się obecnie tylko do prostych przykładów. Oto kilka β-redukcji, redeks dla każdego kroku został podkreślony:

$$\begin{array}{rcl}
\underline{(\lambda x.x)(\lambda y.y)} & \mapsto & \lambda y.y \\
\underline{(\lambda t.\lambda f.t)(\lambda x.x)}(\lambda y.y) & \mapsto & \underline{(\lambda f.\lambda x.x)(\lambda y.y)} \quad \mapsto \quad \lambda x.x \\
\underline{(\lambda t.\lambda f.f)(\lambda x.x)}(\lambda y.y) & \mapsto & \underline{(\lambda f.f)(\lambda y.y)} \quad \mapsto \quad \lambda y.y \\
\underline{(\lambda s.\lambda z.sz)(\lambda x.x)}(\lambda y.y) & \mapsto & \underline{(\lambda z.(\lambda x.x)z)(\lambda y.y)} \quad \mapsto \quad \underline{(\lambda x.x)(\lambda y.y)} \quad \mapsto \quad \lambda y.y
\end{array}$$

β-redukcja jest podstawową zasadą pozwalającą redukować wyrażenia, jednakże nie można z niej otrzymać jednoznacznie wyznaczonych reguł wnioskowania dla osądu $e \mapsto e'$. Innymi słowy, w ogólności może być więcej niż jeden sposób na zastosowanie β-redukcji do danego wyrażenia, lub, równoważnie,

dowolne wyrażenie może zawierać w sobie wiele redeksów. Na przykład, wyrażenie $(\lambda x.x)((\lambda y.y)(\lambda z.z))$ zawiea dwa redeksy:

$$\begin{array}{lllll} \underline{(\lambda x.x)((\lambda y.y)(\lambda z.z))} & \mapsto & \underline{(\lambda y.y)(\lambda z.z)} & \mapsto & \lambda z.z \\ (\lambda x.x)\underline{((\lambda y.y)(\lambda z.z))} & \mapsto & \underline{(\lambda x.x)(\lambda z.z)} & \mapsto & \lambda z.z \end{array}$$

W pierwszym przypadku, wyrażenie które redukujemy jest postaci $(\lambda x.e)e'$ i bezpośrednio stosujemy β-redukcję do całego wyrażenie celem otrzymania $[e'/x]e$. W drugim przypadku, stosujemy β-redukcję do e', które samo jest redeksem; gdyby e' nie było redeksem (na przykład, gdyby $e' = \lambda t.t$), drugi przypadek nie byłby możliwy. Oto jeszcze jeden przykład wyrażenia zawierającego dwa redeksy:

$$\begin{array}{llllll} \underline{(\lambda s.\lambda z.sz)((\lambda x.x)(\lambda y.y))} & \mapsto & \lambda z.\underline{((\lambda x.x)(\lambda y.y))}z & \mapsto & \lambda z.\underline{(\lambda y.y)z} & \mapsto \lambda z.z \\ (\lambda s.\lambda z.sz)\underline{((\lambda x.x)(\lambda y.y))} & \mapsto & \underline{(\lambda s.\lambda z.sz)(\lambda y.y)} & \mapsto & \lambda z.\underline{(\lambda y.y)z} & \mapsto \lambda z.z \end{array}$$

Tym samym w procesie redukowania wyrażenia do wartości możemy być w stanie zastosować β-redukcję na wiele możliwych sposobów. Jako że nie będziemy chcieli stosować β-redukcji tylko w jeden, ściśle określony sposób, będziemy potrzebowali pewnych **strategii redukcyjnych** tak, aby móc usystematyzować zastosowanie β-redukcji.

Skupimy się na dwóch strategiach redukcji: **wywołaniem według nazwy** (*call-by-name*) oraz **wywołaniem według wartości** (*call-by-value*). Z grubsza biorąc, strategia wywołania według nazwy zawsze redukuje redeks położony najbardziej na lewo i na zewnątrz. Mówiąc ściślej, przy danym wyrażeniu e_1e_2, strategia sprawdza, czy e_1 jest λ-abstrakcją $\lambda x.e_1'$. Jeśli tak, to stosuje β-redukcję do całego wyrażenia celem uzyskania $[e_2/x]e_1'$. W przeciwnym razie próbuje zredukować e_1 z zastosowaniem tej samej strategii redukcyjnej bez rozważania e_2; jeżeli w dalszym toku e_1 będzie redukowało się do wartości (która będzie musiała być λ-abstrakcją), strategia zastosuje β-redukcję do całego wyrażenia. W konsekwencji drugie podwyrażenie w aplikacji (to znaczy e_2 w e_1e_2) nigdy nie będzie zredukowane. Strategia wywołania według wartości jest podobna do strategii wywołania według nazwy, ale redukuje drugie podwyrażenie w aplikacji do wartości v po zredukowaniu pierwszego podwyrażenia. Tym samym strategia wywołania według wartości stosuje β-redukcję wyłącznie do aplikacji postaci $(\lambda x.e)v$. Odnotujmy tu, że żadna z opisanych strategii nie redukuje wyrażeń wewnątrz λ-abstrakcji, skąd w szczególności wynika, że wartości nie są w dalszym ciągu redukowane.

Jako przykład rozważmy wyrażenie $(\mathsf{id}_1\mathsf{id}_2)(\mathsf{id}_3(\lambda z.\mathsf{id}_4 z))$, które redukuje się na dwa różne sposoby przy zastosowaniu dwóch różnych strategii redukcyjnych; id_i jest tu skrótem oznaczającym funkcję identycznościową $\lambda x_i.x_i$:

wywołanie według nazwy		wywołanie według wartości
$(\underline{\mathsf{id}_1\mathsf{id}_2})(\mathsf{id}_3(\lambda z.\mathsf{id}_4 z))$		$(\underline{\mathsf{id}_1\mathsf{id}_2})(\mathsf{id}_3(\lambda z.\mathsf{id}_4 z))$
$\mapsto \underline{\mathsf{id}_2(\mathsf{id}_3(\lambda z.\mathsf{id}_4 z))}$		$\mapsto \mathsf{id}_2(\underline{\mathsf{id}_3(\lambda z.\mathsf{id}_4 z)})$
$\mapsto \underline{\mathsf{id}_3(\lambda z.\mathsf{id}_4 z)}$		$\mapsto \underline{\mathsf{id}_2(\lambda z.\mathsf{id}_4 z)}$
$\mapsto \lambda z.\mathsf{id}_4 z$		$\mapsto \lambda z.\mathsf{id}_4 z$

Redukcja "rozjeżdża się" w drugim kroku: strategia przez wywołanie według nazwy używa β-redukcji do całego wyrażenia ponieważ nie musi analizować drugiego podwyrażenia $\mathsf{id}_3(\lambda z.\mathsf{id}_4 z)$, podczas gdy strategia przez wywołanie według wartości wybiera redukcję drugiego podwyrażenia, które jeszcze nie jest wartością.

Możemy teraz wprowadzić reguły wnioskowania dla osądu $e \mapsto e'$, które będziemy nazywali **regułami redukcji**. Strategia wywołania według nazwy używa dwóch reguł redukcji, *Lam* i *App:*

$$\frac{e_1 \mapsto e_1'}{e_1e_2 \mapsto e_1'e_2}\,Lam \qquad \frac{}{(\lambda x.e)e' \mapsto [e'/x]e}\,App$$

Strategia wywołania według wartości wykorzystuje dodatkową regułę do redukowania drugiego podwyrażenia w aplikacjach; dla pozostałych reguł redukcji zapożyczymy takie same nazwy jakie zastosowaliśmy dla strategii wywołania według nazwy:

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}\,Lam \qquad \frac{e_2 \mapsto e_2'}{(\lambda x.e) e_2 \mapsto (\lambda x.e) e_2'}\,Arg \qquad \frac{}{(\lambda x.e) e' \mapsto [e'/x]e}\,App$$

Pewnym minusem strategii wywołania według nazwy jest to, że to samo wyrażenie może być wartościowane wielokrotnie. Na przykład, $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ wartościuje $(\lambda y.y)(\lambda z.z)$ do $\lambda z.z$ dwa razy:

$$\begin{aligned} & \underline{(\lambda x.xx)((\lambda y.y)(\lambda z.z))} \\ \mapsto\ & (\underline{\underline{(\lambda y.y)(\lambda z.z)}})((\lambda y.y)(\lambda z.z)) \\ \mapsto\ & \underline{(\lambda z.z)((\lambda y.y)(\lambda z.z))} \\ \mapsto\ & (\underline{\underline{(\lambda y.y)(\lambda z.z)}}) \\ \mapsto\ & \lambda z.z \end{aligned}$$

W przypadku strategii wywołania według wartości, $(\lambda y.y)(\lambda z.z)$ jest wartościowane tylko raz:

$$\begin{aligned} & (\lambda x.xx)(\underline{(\lambda y.y)(\lambda z.z)}) \\ \mapsto\ & \underline{(\lambda x.xx)(\lambda z.z)} \\ \mapsto\ & \underline{(\lambda z.z)(\lambda z.z)} \\ \mapsto\ & \lambda z.z \end{aligned}$$

Z drugiej strony strategia wywołania według nazwy nigdy nie wartościuje wyrażeń, które nie wnoszą niczego do wartościowań. Na przykład

$$(\lambda t.\lambda f.f)((\lambda y.y)(\lambda z.z))((\lambda y'.y')(\lambda z'.z'))$$

nie wartościuje $(\lambda y.y)(\lambda z.z)$, jako że nie jest ono użyte do wartościowania:

$$\begin{aligned} & \underline{(\lambda t.\lambda f.f)((\lambda y.y)(\lambda z.z))}((\lambda y'.y')(\lambda z'.z')) \\ \mapsto\ & \underline{(\lambda f.f)((\lambda y'.y')(\lambda z'.z'))} \\ \mapsto\ & \ldots \end{aligned}$$

Strategia wywołania według wartości wartościuje $(\lambda y.y)(\lambda z.z)$, ale rezultat $\lambda z.z$ jest ignorowany w kolejnej redukcji:

$$\begin{aligned} & (\lambda t.\lambda f.f)(\underline{(\lambda y.y)(\lambda z.z)})((\lambda y'.y')(\lambda z'.z')) \\ \mapsto\ & \underline{(\lambda t.\lambda f.f)(\lambda z.z)}((\lambda y'.y')(\lambda z'.z')) \\ \mapsto\ & (\lambda f.f)(\underline{(\lambda y'.y')(\lambda z'.z')}) \\ \mapsto\ & \ldots \end{aligned}$$

Strategia wywołania według nazwy jest wykorzystywana przez język funkcyjny Haskell. Haskell jest przykładem "leniwego" języka funkcyjnego, ponieważ wartościuje argumenty do funkcji tylko wtedy, gdy jest to konieczne. W istocie Haskell wykorzystuje inną strategię, zwaną **wywołaniem według potrzeby** (*call-by-need*), która semantycznie jest równoważna wywołaniu według nazwy, ale nigdy nie wartościuje tego samego wyrażenia więcej, niż jeden raz. Strategia wywołania według wartości jest z kolei zaimplementowana w SML-u.

Powiemy, że wyrażenie jest w **postaci normalnej**, gdy nie stosuje się do niego żadna reguła redukcji. Oczywiście każda wartość (która, w przypadku rachunku λ, jest λ-abstrakcją) jest w postaci normalnej.

Istnieją wszelako wyrażenia w postaci normalnej, które nie są wartościami. Na przykład, $x\lambda y.y$ jest w postaci normalnej, albowiem x nie może być dalej zredukowane, ale nie jest też wartością. Będziemy mówili, że takie wyrażenie – albo jego redukcja – **zatyka się**. O wyrażeniu, które się zatyka, możemy myśleć jak o wadliwym programie, który nie powinien pojawić się w procesie wartościowania. W dalszym ciągu wykładu omówimy rozszerzenie rachunku λ, tzw. **rachunek λ z typami prostymi**, który statycznie (a więc w czasie kompilacji) gwarantuje, że program spełniający pewne określone warunki nigdy się nie zatka.

5.3. **Podstawianie.** Zanim przejdziemy do definicji podstawiania w rachunku λ, będziemy potrzebowali definicji **zmiennych wolnych,** które z kolei są dokładną przeciwnością zmiennych związanych. Zmienna wolna to zmienna, która nie jest związana w żadnej obejmującej ją λ- abstrakcji. Na przykład, y w $\lambda x.y$ jest zmienną wolną, ponieważ żadna z λ-abstrakcji postaci $\lambda y.e$ jej nie wiąże. Aby formalnie zdefiniować zmienne wolne, wprowadzimy indukcyjnie pojęcie zbioru $FV(e)$ wszystkich zmiennych związanych w e:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x.e) & = & FV(e) \setminus \{x\} \\ FV(e_1 e_2) & = & FV(e_1) \cup FV(e_2) \end{array}$$

Jako że każda zmienna jest albo wolna, albo związana, zmienna x w e taka, że $x \notin FV(e)$ musi być związana w pewnej λ-abstrakcji. Powiemy, że wyrażenie e jest **domknięte**, jeśli nie zawiera zmiennych wolnych, czyli jeśli $FV(e) = \emptyset$. Oto kilka przykładów:

$$\begin{array}{rcl} FV(\lambda x.x) & = & \{\} \\ FV(xy) & = & \{x, y\} \\ FV(\lambda x.xy) & = & \{y\} \\ FV(\lambda y.\lambda x.xy) & = & \{\} \\ FV((\lambda x.xy)(\lambda x.xz)) & = & \{y, z\} \end{array}$$

Podstawianie $[e/x]e'$ jest zdefiniowane indukcyjnie poprzez następujące przypadki:

$$\begin{array}{rcll} [e/x]x & = & e & \\ [e/x]y & = & y & \text{jeśli } x \neq y \\ [e/x](e_1 e_2) & = & [e/x]e_1[e/x]e_2. & \end{array}$$

Chcąc podać precyzyjną definicję podstawiania dla pozostałego przypadku $[e/x]\lambda y.e'$, musimy najpierw zrozumieć dwie własności zmiennych. Pierwsza z nich jest taka, że nazwa zmiennej związanej nie ma znaczenia, co zresztą odpowiada naszej intuicji. Na przykład funkcja identycznościowa $\lambda x.x$ wewnątrz wyrażenia e może być przepisana jako $\lambda y.y$ dla dowolnej zmiennej y bez zmiany znaczenia e, jako że zarówno $\lambda x.x$ jak i $\lambda y.y$ oznaczają funkcję identycznościową. Jako inny przykład możemy przepisać $\lambda x.\lambda y.xy$ jako $\lambda y.\lambda x.yx$, gdzie obydwa wyrażenia oznaczają taką samą funkcję, która stosuje pierwszy argument do drugiego.

Formalnie używamy osądu $e \equiv_\alpha e'$ aby powiedzieć, że e może być zapisane jako e' przez zmianę nazw zmiennych związanych w e i vice versa. Oto kilka przykładów $e \equiv_\alpha e'$:

$$\begin{array}{rcl} \lambda x.x & \equiv_\alpha & \lambda y.y \\ \lambda x.\lambda y.xy & \equiv_\alpha & \lambda z.\lambda y.zy \\ \lambda x.\lambda y.xy & \equiv_\alpha & \lambda x.\lambda z.xz \\ \lambda x.\lambda y.xy & \equiv_\alpha & \lambda y.\lambda x.yx \end{array}$$

Relację $\equiv_\alpha$ będziemy nazwać **α-równoważnością**. Będziemy też mówić, że **α-konwersji** e w e' przepisuje e jako e' przez zmianę nazw zmiennych związanych w e.

Omówiona przez nas pierwsza własność uzasadnia wprowadzenie następującego fragmentu definicji podstawiania:

$$[e'/x]\lambda x.e = \lambda x.e$$

Intuicyjnie, jeśli przepiszemy $\lambda x.e$ jako inną λ-abstrakcję postaci $\lambda y.e''$, gdzie y jest nową zmienną taką, że $x \neq y$, to podstawienie $[e'/x]$ będze zignorowane, ponieważ nigdzie w $\lambda y.e''$ nie występuje x. Oto prosty przykład z $e = x$:

$$\begin{array}{rcl} [e'/x]\lambda x.x & \equiv_\alpha & [e'/x]\lambda y.y \\ & = & \lambda y.y \\ & \equiv_\alpha & \lambda x.x \end{array}$$

Uogólnienie tego przypadku jest takie, że $[e'/x]$ nie ma wpływu na e o ile x nie jest zmienną wolną w e:

$$[e'/x]e = e \text{ jeśli } x \notin FV(e).$$

Innymi słowy, będziemy chcieli stosować $[e'/x]$ do e tylko wtedy, gdy x jest zmienną wolną w e.

Druga z własności, na które musimy zwrócić uwagę jest taka, że zmienna wolna x w wyrażeniu e nigdy nie stanie się zmienną związaną; jeśli *explicite* zastąpimy ją przez inne wyrażenie e', tak jak w $[e'/x]e$, po prostu zniknie. Aby lepiej zrozumieć tę własność, rozważmy naiwną definicję podstawienia $[e'/x]\lambda y.e$, gdzie y może, ale nie musi być zmienną wolną w e':

$$[e'/x]\lambda y.e = \lambda y.[e'/x]e \text{ jeśli } x \neq y.$$

Teraz jeśli y jest zmienną wolną w e', automatycznie staje się zmienną związaną w $\lambda y.[e'/x]e$, co nie jest dopuszczalne. Oto przykład ukazujący taką anomalię:

$$(\lambda x.\lambda y.x)y \mapsto [y/x]\lambda y.x = \lambda y.[y/x]x = \lambda y.y$$

Przed podstawieniem $\lambda y.x$ jest λ-abstrakcją która ignoruje swój argument i zwraca x, ale po podstawieniu zamienia się w funkcję identycznościową! W przykładzie tym zmienna wolna y, która ma być podstawiona za x powinna dalej być wolna po podstawieniu, ale przypadkowo zostaje **przechwycona** przez λ-abstrakcję $\lambda y.x$ i staje się zmienną związaną. Tego typu zjawisko będziemy nazywali **przechwytywaniem zmiennych,** które kłóci się z naturalną intuicją, że zmienna wolna pozostaje wolna, o ile nie zostanie zastąpiona przez inne wyrażenie. Ta obserwacja uogólnia się do następującej definicji $[e'/x]\lambda y.e$, którą nazwiemy **podstawieniem unikającym przechwytywania zmiennych:**

$$[e'/x]\lambda y.e = \lambda y.[e'/x]e \text{ jeśli } x \neq y, y \notin FV(e').$$

Jeśli nastąpi przechwycenie zmiennej, ponieważ $y \in FV(e')$, zmieniamy nazwę y na inną zmienną, która nie jest wolna w e. Na przykład, $(\lambda x.\lambda y.xy)y$ można bezpiecznie zredukować po uprzedniej zmianie nazwy zmiennej związanej y na nową zmienną z:

$$(\lambda x.\lambda y.xy)y \mapsto [y/x]\lambda y.xy \equiv_\alpha [y/x]\lambda z.xz = \lambda z.yz.$$

Podamy teraz definicję osądu $e \equiv_\alpha e'$. Potrzebować będziemy w tym celu pojęcia **wymiany zmiennych** $[x \leftrightarrow y]e$, która polega na zamianie *wszystkich* wystąpień x w e na y i *wszystkich* wystąpień y w e na x. Podkreślmy raz jeszcze, że wymieniamy wszystkie wystąpienia zmiennych, nawet te bezpośrednio po λ w λ-abstrakcjach, co zresztą czyni implementację $[x \leftrightarrow y]e$ dość bezpośrednią. Oto przykład: $[x \leftrightarrow y]\lambda x.\lambda y.xy = \lambda y.[x \leftrightarrow y]\lambda y.xy = \lambda y.\lambda x.[x \leftrightarrow y](xy) = \lambda y.\lambda x.yx$.

Definicja $e \equiv_\alpha e'$ jest dana indukcyjnie poprzez następujące reguły:

$$\frac{}{x \equiv_\alpha x'} Var_\alpha \qquad \frac{e_1 \equiv_\alpha e_1' \qquad e_2 \equiv_\alpha e_2'}{e_1 e_2 \equiv_\alpha e_1' e_2'} App_\alpha$$

$$\frac{e \equiv_\alpha e'}{\lambda x.e \equiv_\alpha \lambda x.e'} Lam_\alpha \qquad \frac{x \neq y \qquad y \notin FV(e) \qquad [x \leftrightarrow y]e \equiv_\alpha e'}{\lambda x.e \equiv_\alpha \lambda y.e'} Lam'_\alpha$$

Reguła Lam_α powiada, że aby porównać $\lambda x.e$ z $\lambda x.e'$, które wiążą tę samą zmienną, porównujemy e z e'. Aby porównać dwie λ-abstrakcje wiążące różne zmienne, stosujemy regułę Lam'_α.

Chcąc zobaczyć, dlaczego reguła Lam'_α działa, musimy zrozumieć, co pociąga za sobą przesłanka $y \notin FV(e)$. Jako że $y \notin FV(e)$ pociąga $y \notin FV(\lambda x.e)$ i oczywiście $x \notin FV(\lambda x.e)$, zewnętrzny obserwator nie zauważy różnicy nawet wtedy, gdy zmienne x i y zostałyby dosłownie wymienione wewnątrz $\lambda x.e$. Innymi słowy $\lambda x.e$ oraz $[x \to y]\lambda x.e$ nie różnią się z punktu widzenia zewnętrznego obserwatora. Ponieważ $[x \to y]\lambda x.e = \lambda y.[x \to y]e$, porównujemy $[x \to y]e$ z e', co jest dokładnie treścią trzeciej przesłanki w regule Lam'_α. Jako przykład podamy dowód $\lambda x.\lambda y.xy \equiv_\alpha \lambda y.\lambda x.yx$:

$$\dfrac{x \neq y \qquad y \notin FV(\lambda y.xy) \qquad \dfrac{\dfrac{}{y \equiv_\alpha y} Var_\alpha \qquad \dfrac{}{x \equiv_\alpha x} Var_\alpha}{yx \equiv_\alpha yx} App_\alpha}{\lambda x.yx \equiv_\alpha \lambda x.yx} Lam_\alpha}{\lambda x.\lambda y.xy \equiv_\alpha \lambda y.\lambda x.yx} Lam_\alpha$$

Zadanie 9. *Czy można udowodnić, że $\lambda x.e \equiv_\alpha \lambda y.e'$, jeśli $x \neq y$ oraz $y \in FV(e)$?*

Zadanie 10. *Załóżmy, że $x \notin FV(e)$ oraz $y \notin FV(e)$. Udowodnij, że $e \equiv_\alpha [x \leftrightarrow y]e$.*

Możemy teraz podać kompletną definicję podstawiania:

$$\begin{array}{rcll}
[e/x]x & = & e & \\
[e/x]y & = & y & \text{jeśli } x \neq y \\
[e/x](e_1 e_2) & = & [e/x]e_1[e/x]e_2. & \\
[e'/x]\lambda x.e & = & \lambda x.e & \\
[e'/x]\lambda y.e & = & \lambda y.[e'/x]e & \text{jeśli } x \neq y, y \notin FV(e') \\
[e'/x]\lambda y.e & = & \lambda z.[e'/x][y \leftrightarrow z]e & \text{jeśli } x \neq y, y \in FV(e') \\
 & & & \text{gdzie } z \neq y, z \notin FV(e), z \neq x, z \notin FV(e').
\end{array}$$

Ostatnie równanie pociąga w szczególności, że jeśli y jest zmienną wolną w e', możemy wybrać inną zmienną z spełniającą warunki podane po słowie *gdzie* i przepisać $\lambda y.e$ jako $\lambda z.[y \leftrightarrow z]e$ zgodnie z następującą α-konwersją:

$$\frac{y \neq z \qquad z \notin FV(e) \qquad [y \leftrightarrow z]e \equiv_\alpha [y \leftrightarrow z]e}{\lambda y.e \equiv_\alpha \lambda z.[y \leftrightarrow z]e} Lam'_\alpha$$

Założenie $z \notin FV(e')$ pozwala przepisać $[e'/x]\lambda z.[y \leftrightarrow z]e$ jako $\lambda z.[e'/x][y \leftrightarrow z]e$. W większości przypadków uzyskujemy zmienną z po prostu przez wygenerowanie nowej zmiennej. W takim wypadku nigdy nie zamieniamy z przez y i tym samym ostatnie równanie można przepisać jako:

$$[e'/x]\lambda y.e = \lambda z.[e'/x][z/y]e.$$

Tym nie mniej tak zapisane równanie jest mniej wydajne. Rozważmy na przykład $e = xy(\lambda y.xy)$. $[y \leftrightarrow z]e$ daje $xz(\lambda z.xz)$ i $[e'/x][y \leftrightarrow z]e$ nie przechwytuje zmiennej:

$$[e'/x][y \leftrightarrow z]e = [e'/x](xz(\lambda z.xz)) = e'z(\lambda z.e'z).$$

Dla kontrastu, $[z/y]e$ daje $xz(\lambda y.xy)$ i $[e'/x][z/y]e$ przechwytuje zmienną w $[e'/x]\lambda y.xy$:

$$[e'/x][z/y]e = [e'/x](xz(\lambda y.xy)) = e'z[e'/x](\lambda y.xy).$$

Tym samym musimy wygenerować jeszcze jedną nową zmienną.

Zadanie 11. *Dokończyć α-konwersje w podanych niżej przykładach, w których dokonano już wyboru nowej zmiennej:*

(1) $\lambda x.\lambda x'.xx' \equiv_\alpha \lambda x'.\ldots$.
(2) $\lambda x.\lambda x'.xx'x'' \equiv_\alpha \lambda x'.\ldots$.
(3) $\lambda x.\lambda x'.xx'x'' \equiv_\alpha \lambda x''.\ldots$.